Abstract

IP Multimedia Subsystem (IMS) is an architectural framework for delivering Internet protocol multimedia to mobile users. The functionality of the system is specified and standardised to allow interoperability between equipment from different vendors. We study how to build the system models for model-based testing of an IMS configuration in a way which provides easy means of linking a requirement, i.e. some paragraph in the specification to a specific part of the model. In addition, it allows to find out which requirements listed in the specification are violated in case of an error. Requirements-based approach in testing is extensively used in the industry in manual and script-based testing. The same should be possible for model-based testing. The current case study describes how to model a fragment of the IMS protocols from the point of view of the requirements and provides solutions using either a network of Uppaal automata or NModel model programs.

1. Introduction

We present a case study of applying requirements-driven modelling and testing on a fragment of the IP Multimedia Subsystem (IMS) used in the telecommunication industry. It was originally designed by the wireless standards body 3rd Generation Partnership Project (3GPP), and is part of the vision for evolving mobile networks beyond GSM. IMS interoperability tests standardized by ETSI are produced to facilitate interoperability of IMS networks from different vendors. In the current example there are long lists of detailed requirements that need to be satisfied by different components that constitute the system.

Our goal is to model the system and to apply model-driven testing with the objective to achieve full requirements coverage. For testing the compatibility of the implementations with the specification it is important to get feedback which requirements are implemented correctly by the implementation and which are violated.

We use two different techniques for modelling the IMS case study but both approaches serve the same purpose and are driven by similar ideas. We use a network of Uppaal automata and NModel model programs for formalising the specifications.

Our goal is to keep the model as close as possible to the specification document in a way where components of the model are traceable to paragraphs in the specification. The formalisation is done in two steps: formalising the infrastructure or the configuration of the system and formalising separate requirements or features as separate components that are connected to the rest of the model by composition at run time.

As the word requirement has a variety of meanings we use the definition given in [4]: A requirement is a testable statement of some functionality that the product must have.

We have chosen Uppaal to build extended finite state machine (EFSM) models as Uppaal supports automatic composition of automata, has a model checking backend for analysing the model and generating tests, and has a nice GUI. NModel on the other hand supports building model programs and splitting them up into features that are designed for requirement-oriented modelling. An important aspect in those formalisms is that they support composition. We do not compare the tools but show that the requirement-oriented approach can be used in different settings.

2. IMS Case-study

The proposed technique is illustrated by the case-study proposed by ETSI[4] and focuses on interoperability testing between different IMS networks. The IMS

Figure 1: Test configuration for the interoperability testing of the IMS network.

is an architectural framework for delivering Internet protocol multimedia to fixed as well as mobile users. IMS is based on the Session Initiation Protocol (SIP).

We use a minimal network configuration of IMS that consists of Proxy Call Session Control Function (P-CSCF), i(terrogating)-CSCF, S(erving)-CSCF, and Home Subscriber Server (HSS) components. For the purpose of testing we do not distinguish the individual components listed above. Instead we look at the system from the perspective of the ports where we can control and observe messages.

As shown in the Figure 1 the system under test (SUT) consists of two internetworking core IMS networks. They exchange SIP messages over the channel Mw. Both IMS networks have a user equipment (UE) registered to the network. The UEs are the main test drivers, being part of the interoperability test system. Thus the messages on the Gm interfaces can be generated and observed by the testing system. The messages on the Mw channel can only be observed.

The usual test development goes through the stepwise test development procedure including the requirements identification, requirements cataloguing, test purpose identification for each requirement following by the development of the test descriptions and executable test cases. The purpose of the case study is the investigation of the possibility of generating the test cases from the model of the system and requirements.

We will model the following the IMS specification [5]. In particular part of the requirement 5.1.1.7:

In the case where the S-CSCF has knowledge that the SIP URI contained in the received P-Asserted-Identity header is an alias SIP URI for a tel URI, the S-CSCF shall add a second P-Asserted-Identity header containing this tel URI.

3. Modelling with a Network of Automata

We demonstrate how a network of EFMSs, in our case Uppaal automata [2], can be used for modelling the SUT and the requirements. The tests can be generated from the traces obtained by reachability analysis and strongest postcondition calculation. We use Uppaal automata as the modelling language and Uppaal model checker as the reachability analysis tool, but the method is not specific to Uppaal, but an alternative should allow the use of data structures, communication channels and/or shared variables, synchronous composition, and reachability analysis of the model.

The composed system model consists of the model of the SUT, models of the requirements and the test data generator. An example of the model are shown on the Figure 2.

The model of the SUT specifies the general structure of the components and channels of the system. The components of the model have some abstract “synchronization points” where the automata representing some specific requirements can hook to and mark the corresponding behaviours in the actual system. The model is kept as abstract as possible to match the textual specifications it models. Only the details appearing in the requirements are modelled. It is possible that the model of the SUT is refined for some specific requirement, thus resulting in several different models of the SUT. The model of the SUT can include some general knowledge about the behaviour of the system, e.g. “any component must see a response to any message it forwards”, but actual restrictions on all possible behaviours stem from specific requirements. The automaton IMS-X on the Figure 2 is a template for the IMS component of SUT.

The model of a requirement consists of one or several automata synchronizing with the model of the SUT at some synchronization points. The requirement automata synchronize in an unobtrusive manner - they synchronize only when the conditions of the requirement are satisfied at the synchronization point and

$$
\text{is_sip_uri(MIMSB.c[P\_Asserted\_Identity\_header])}
\land
MIMSB.c[P\_Asserted\_Identity\_header] ==
RegB.c[SIP]

\text{MIMSB.c[P\_Asserted\_Identity\_header2] = RegB.c[TEL]}
$$

Figure 2: Components of the model
don’t restrict the behaviour if not, thus “labeling” the satisfying behaviours. The automaton Req5117-1 in the Figure 2c formalises the corresponding requirement.

The test data generator is a nondeterministic source of data. The modeller should provide enough data to allow covering all requirements. Whether the variety of data is sufficient can be determined by reachability analysis. It is not advisable to link the test data to a specific requirement, but let the system choose the data what allows the requirement to be satisfied. The automaton Messenger in the Figure 2c is a simple data generator.

3.1. Test Sequence Generation

Test can be generated from a trace of a run of the network of automata. Trace is an instance of the behaviour satisfying some property. We are interested in traces where the automaton of the requirement of interest is in at the location Sat and the automata of the SUT are are at the locations Ready. It can be formulated as a reachability problem \( \exists \diamond Req5117_1.Sat \land UE_A.Ready \land UE_B.Ready \land IMS_A.Ready \land IMS_B.Ready \). Such reachability problem can be solved and a trace generated by a model checker or other reachability analysis tool. The trace consists of sequence of state transitions taken by the automata including the synchronization events and variable assignments. For test generation we generate a condition for the content of the message on all visible communication points using the strongest postcondition analysis on the generated trace and writing out the postcondition at the communication points. The conditions before the first internal communication of the SUT constitute the pre-test conditions and the rest form the test pass criteria.

Every requirement or subrequirement is expressed as a separate automaton and a specific test or set of tests can be generated for each. The combined tests for checking many requirements in the same run can be generated by finding a trace where all the requirement automata end up in the state Sat. If such a trace does not exist, it can be concluded that some requirements cannot be tested together. It may happen due to incompatible pre-conditions on data. The search for the groups of requirements what can be tested together can be done by an automated procedure.

4. Modelling with Model Programs

Model programs gives a textual alternative for modelling. The concept is used in industrial tools, such as SpecExplorer [6]. Model programs are unwound into transition systems that can be used in model-based testing, for conformance checking of a SUT and for design validation.

A recent toolkit NModel [3], [7] provides a library and tools for building models using C#. A feature specific toNModel is that it facilitates composition of model programs [8]. This allows splitting the model up into features corresponding to either the configuration of the infrastructure or specific requirements.

The structure of the IMS and UE components of the model can be specified in an object-oriented way as in Figure 3. Model programs make use of NModel library utilities, like LabeledInstance for specifying for the engine that the IMS and UE are object structures, and data structures, but also C# enumerations.

A model program consists of global data structures, actions and action guards. In Figure 4 there is an action IMSReceiveResp which denotes the reception of a response message when it is received by the IMS. The action has two guards which are boolean functions with Enabled added to the name of the action. The action body has been left empty deliberately because the behaviour of the IMS upon reception of certain types of messages is specified in separate requirement models. The action has more header arguments which have left been left out for the sake of brevity.

```csharp
public enum UEName { A, B }

public class UE : LabeledInstance<UE>
{
    // name needs to be set manually before the instance can be used.
    public UEName name;
    // the ims to which the UE is connected
    public IMS ims;
    public Set<SIPMessage> receivedMessages;

    public override void Initialize()
    {
        receivedMessages = Set<SIPMessage>.EmptySet;
    }
}

public class IMS : LabeledInstance<IMS>
{
    public IMS ims;
    public Set<SIPMessage> msgBuffer;
    public Set<SIPMessage> responseBuffer;

    public override void Initialize()
    {
        msgBuffer = Set<SIPMessage>.EmptySet;
        responseBuffer = Set<SIPMessage>.EmptySet;
    }
}
```

Figure 3: Structure of the network components IMS and UE.

In Figure 5 there is a feature corresponding to the requirement mentioned above. As the requirement
[Action]
public static void IMSReceiveResp(  
    [Domain("imss")]
    IMS ims,  
    [Domain("IMSResponses")]
    ResponseID msgid,  
    PreferredIdentityHeader pih,  
    AssertedIdentityHeader aih,  
    ViaHeader vh,  
    CalledPartyIDHeader cpidh,  
    ChargingVectorHeader cvh)
{
}
public static bool IMSReceiveRespEnabled(  
    IMS ims, ResponseID msgid)
{
    return ims.responseBuffer.Contains(m);
}

Figure 4: An action at the port between IMS and UE denoting the situation where the IMS receives a response message from the UE.

[Feature("R5117-1")]
public static class R5117
{
    [Requirement("If the P-Asserted-Identity header contains ...")]
    [Action("IMSReceiveResp(ims,msgid,_,aih,_,_,_)")]
    public static void ReceiveResp(  
        IMS ims, ResponseID msgid,  
        AssertedIdentityHeader aih)
    {
        if (aih.Contains(URISort.TEL)  
            && !aih.Contains(URISort.SIP))
            aih = aih.Add(URISort.SIP);
    /* code to update the global state variables  
     * corresponding to the appropriate headers of message msgid 
     */
    }
    public static bool ReceiveRespEnabled(IMS ims,  
        ResponseID msgid,  
        AssertedIdentityHeader aih)
    {
        return aih.Count > 0; // not empty
    }
}

Figure 5: A feature corresponding to the requirement 5.1.1.7.

mentions only P-Asserted-Identity header, we can ignore the rest of the arguments of the action. The NModel framework provides means for composing such features into a model and generating test sequences or running online tests to check their validity in the system.

5. Conclusion

We presented the results of applying requirements-driven model-based testing to a fragment of the IP Multimedia subsystem. We modelled the system with two different formalisms — network of Uppaal automata and NModel model programs. We learned that by using appropriate composition, it is possible to present separate requirements as separate automata or features in the case of model programs. Such requirement automata or features can be appropriately composed by the model of the infrastructure that connects the physical components of the system. The requirements-drivenness is due to our objective to cover all requirements and the way the models are built: the models are partitioned into the common infrastructure and separate components corresponding to concrete requirements as precisely as allowed by the modelling formalism.

After building the models the test engineer can select which requirements are to be tested and to generate corresponding test cases from a composition of the infrastructure model and the appropriate requirement models. It is also possible to apply similar approach to online testing of a selection of requirements at a time.

Another outcome of the current work is the need for tool support for selecting which requirements to compose into the system at any given time as there are interdependencies between different requirements.

Acknowledgments

The authors wish to thank Stephan Schultz from ETSI for providing the specifications of the IMS case study. The authors would also like to thank the anonymous referees for valuable comments. The work was partially supported by the ITEA-2 D-MINT project.

References